AVF Control Number: AVF-VSR-174.1088
88-03-25-VRX

Ada COMPILER VALIDATION SUMMARY REPORT:

Certificate Number: 880606W1.09083
Verdix Corporation
VAda-110-03406, Version V5.51
DEC MicroVAX II Host to Fairchild 9450 Target

Completion of On-Site Testing:
13 June 1988

Prepared By:
Ada Validation Facility
ASD/SCEL
Wright-Patterson AFB OH 45433-6503

Prepared For:
Ada Joint Program Office
United States Department of Defense
Washington DC 20301-3081

UNCLASSIFIED
SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered) AD-A204 424

REPORT DOCUMENTATION PAGE	READ INSTRUCTIONS BEFORE COMPLETEING FORM
1. REPORT NUMBER | 2. GOVT ACCESSION NO.	3. RECIPIENT'S CATALOG NUMBER
4. TITLE (and Subtitle) Ada Compiler Validation Summary Report: Verdix Corporation, VAda-110-03406, Version V5.51, DEC MicroVAX II (Host) to Fairchild 9450 (Target). (880606W1.09083)	5. TYPE OF REPORT & PERIOD COVERED 13 June 1988 to 13 June 1989
	6. PERFORMING ORG. REPORT NUMBER
7. AUTHOR(s) Wright-Patterson Air Force Base, Dayton, Ohio, U.S.A.	8. CONTRACT OR GRANT NUMBER(s)
9. PERFORMING ORGANIZATION AND ADDRESS Wright-Patterson Air Force Base, Dayton, Ohio, U.S.A.	10. PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS
11. CONTROLLING OFFICE NAME AND ADDRESS Ada Joint Program Office United States Department of Defense Washington, DC 20301-3081	12. REPORT DATE 13 June 1988
	13. NUMBER OF PAGES 40 p.
14. MONITORING AGENCY NAME & ADDRESS (If different from Controlling Office) Wright-Patterson Air Force Base, Dayton, Ohio, U.S.A.	15. SECURITY CLASS (of this report) UNCLASSIFIED
	15a. DECLASSIFICATION/DOWNGRADING SCHEDULE N/A

16. DISTRIBUTION STATEMENT (of this Report)

Approved for public release; distribution unlimited.

17. DISTRIBUTION STATEMENT (of the abstract entered in Block 20. If different from Report)

UNCLASSIFIED

18. SUPPLEMENTARY NOTES

19. KEYWORDS (Continue on reverse side if necessary and identify by block number)

Ada Programming language, Ada Compiler Validation Summary Report, Ada Compiler Validation Capability, ACVC, Validation Testing, Ada Validation Office, AVO, Ada Validation Facility, AVF, ANSI/MIL-STD-1815A, Ada Joint Program Office, AJPO

20. ABSTRACT (Continue on reverse side if necessary and identify by block number)

VAda -110-03406, Version V5.51, Verdix Corporation, Wright-Patterson Air Force Base, DEC MicroVAX II under VMS, Version 4.7 (Host) to Fairchild 9450 under Tektronix emulation (Target), ACVC 1.9.

Ada Compiler Validation Summary Report:

Compiler Name: VAda-110-03406, Version V5.51

Certificate Number: 880606W1.09083

Host:	Target:
DEC MicroVAX II under VMS, Version 4.7	Fairchild 9450 under Tektronix emulation

Testing Completed 13 June 1988 Using ACVC 1.9

This report has been reviewed and is approved.

[illegible]

Ada Validation Facility
Steven P. Wilson
Technical Director
ASD/SCEL
Wright-Patterson AFB OH 45433-6503

[illegible]

Ada Validation Organization
Dr. John F. Kramer
Institute for Defense Analyses
Alexandria VA 22311

Virginia L. Castor

Ada Joint Program Office
Virginia L. Castor
Director
Department of Defense
Washington DC 20301

TABLE OF CONTENTS

CHAPTER 1

INTRODUCTION

This Validation Summary Report ~~(VSR)~~ describes the extent to which a specific Ada compiler conforms to the Ada Standard, ANSI/MIL-STD-1815A. This report explains all technical terms used within it and thoroughly reports the results of testing this compiler using the Ada Compiler Validation Capability ~~(ACVC)~~. An Ada compiler must be implemented according to the Ada Standard, and any implementation-dependent features must conform to the requirements of the Ada Standard. The Ada Standard must be implemented in its entirety, and nothing can be implemented that is not in the Standard.

Even though all validated Ada compilers conform to the Ada Standard, it must be understood that some differences do exist between implementations. The Ada Standard permits some implementation dependencies--for example, the maximum length of identifiers or the maximum values of integer types. Other differences between compilers result from the characteristics of particular operating systems, hardware, or implementation strategies. All the dependencies observed during the process of testing this compiler are given in this report.

The information in this report is derived from the test results produced during validation testing. The validation process includes submitting a suite of standardized tests, the ACVC, as inputs to an Ada compiler and evaluating the results. The purpose of validating is to ensure conformity of the compiler to the Ada Standard by testing that the compiler properly implements legal language constructs and that it identifies and rejects illegal language constructs. The testing also identifies behavior that is implementation dependent but permitted by the Ada Standard. Six classes of tests are used. These tests are designed to perform checks at compile time, at link time, and during execution.

(KR) (

1.1 PURPOSE OF THIS VALIDATION SUMMARY REPORT

This VSR documents the results of the validation testing performed on an Ada compiler. Testing was carried out for the following purposes:

- To attempt to identify any language constructs supported by the compiler that do not conform to the Ada Standard
- To attempt to identify any language constructs not supported by the compiler but required by the Ada Standard
- To determine that the implementation-dependent behavior is allowed by the Ada Standard

Testing of this compiler was conducted by SofTech, Inc. under the direction of the AVF according to procedures established by the Ada Joint Program Office and administered by the Ada Validation Organization (AVO). On-site testing was completed 13 June 1988 at Aloha, OR.

1.2 USE OF THIS VALIDATION SUMMARY REPORT

Consistent with the national laws of the originating country, the AVO may make full and free public disclosure of this report. In the United States, this is provided in accordance with the "Freedom of Information Act" (5 U.S.C. #552). The results of this validation apply only to the computers, operating systems, and compiler versions identified in this report.

The organizations represented on the signature page of this report do not represent or warrant that all statements set forth in this report are accurate and complete, or that the subject compiler has no nonconformities to the Ada Standard other than those presented. Copies of this report are available to the public from:

Ada Information Clearinghouse
Ada Joint Program Office
OUSDRE
The Pentagon, Rm 3D-139 (Fern Street)
Washington DC 20301-3081

or from:

Ada Validation Facility
ASD/SCEL
Wright-Patterson AFB OH 45433-6503

Questions regarding this report or the validation test results should be directed to the AVF listed above or to:

> Ada Validation Organization
> Institute for Defense Analyses
> 1801 North Beauregard Street
> Alexandria VA 22311

1.3 REFERENCES

1. Reference Manual for the Ada Programming Language, ANSI/MIL-STD-1815A, February 1983 and ISO 8652-1987.

2. Ada Compiler Validation Procedures and Guidelines, Ada Joint Program Office, 1 January 1987.

3. Ada Compiler Validation Capability Implementers' Guide, SofTech, Inc., December 1986.

4. Ada Compiler Validation Capability User's Guide, December 1986.

1.4 DEFINITION OF TERMS

ACVC	The Ada Compiler Validation Capability. The set of Ada programs that tests the conformity of an Ada compiler to the Ada programming language.
Ada Commentary	An Ada Commentary contains all information relevant to the point addressed by a comment on the Ada Standard. These comments are given a unique identification number having the form AI-ddddd.
Ada Standard	ANSI/MIL-STD-1815A, February 1983 and ISO 8652-1987.
Applicant	The agency requesting validation.
AVF	The Ada Validation Facility. The AVF is responsible for conducting compiler validations according to procedures contained in the Ada Compiler Validation Procedures and Guidelines.
AVO	The Ada Validation Organization. The AVO has oversight authority over all AVF practices for the purpose of maintaining a uniform process for validation of Ada compilers. The AVO provides administrative and technical

	support for Ada validations to ensure consistent practices.
Compiler	A processor for the Ada language. In the context of this report, a compiler is any language processor, including cross-compilers, translators, and interpreters.
Failed test	An ACVC test for which the compiler generates a result that demonstrates nonconformity to the Ada Standard.
Host	The computer on which the compiler resides.
Inapplicable test	An ACVC test that uses features of the language that a compiler is not required to support or may legitimately support in a way other than the one expected by the test.
Passed test	An ACVC test for which a compiler generates the expected result.
Target	The computer for which a compiler generates code.
Test	A program that checks a compiler's conformity regarding a particular feature or a combination of features to the Ada Standard. In the context of this report, the term is used to designate a single test, which may comprise one or more files.
Withdrawn test	An ACVC test found to be incorrect and not used to check conformity to the Ada Standard. A test may be incorrect because it has an invalid test objective, fails to meet its test objective, or contains illegal or erroneous use of the language.

1.5 ACVC TEST CLASSES

Conformity to the Ada Standard is measured using the ACVC. The ACVC contains both legal and illegal Ada programs structured into six test classes: A, B, C, D, E, and L. The first letter of a test name identifies the class to which it belongs. Class A, C, D, and E tests are executable, and special program units are used to report their results during execution. Class B tests are expected to produce compilation errors. Class L tests are expected to produce compilation or link errors.

Class A tests check that legal Ada programs can be successfully compiled and executed. There are no explicit program components in a Class A test to check semantics. For example, a Class A test checks that reserved words of another language (other than those already reserved in the Ada language) are not treated as reserved words by an Ada compiler. A Class A test is passed if no errors are detected at compile time and the program executes to produce a PASSED message.

Class B tests check that a compiler detects illegal language usage. Class B tests are not executable. Each test in this class is compiled and the resulting compilation listing is examined to verify that every syntax or semantic error in the test is detected. A Class B test is passed if every illegal construct that it contains is detected by the compiler.

Class C tests check that legal Ada programs can be correctly compiled and executed. Each Class C test is self-checking and produces a PASSED, FAILED, or NOT APPLICABLE message indicating the result when it is executed.

Class D tests check the compilation and execution capacities of a compiler. Since there are no capacity requirements placed on a compiler by the Ada Standard for some parameters--for example, the number of identifiers permitted in a compilation or the number of units in a library--a compiler may refuse to compile a Class D test and still be a conforming compiler. Therefore, if a Class D test fails to compile because the capacity of the compiler is exceeded, the test is classified as inapplicable. If a Class D test compiles successfully, it is self-checking and produces a PASSED or FAILED message during execution.

Each Class E test is self-checking and produces a NOT APPLICABLE, PASSED, or FAILED message when it is compiled and executed. However, the Ada Standard permits an implementation to reject programs containing some features addressed by Class E tests during compilation. Therefore, a Class E test is passed by a compiler if it is compiled successfully and executes to produce a PASSED message, or if it is rejected by the compiler for an allowable reason.

Class L tests check that incomplete or illegal Ada programs involving multiple, separately compiled units are detected and not allowed to execute. Class L tests are compiled separately and execution is attempted. A Class L test passes if it is rejected at link time--that is, an attempt to execute the main program must generate an error message before any declarations in the main program or any units referenced by the main program are elaborated.

Two library units, the package REPORT and the procedure CHECK_FILE, support the self-checking features of the executable tests. The package REPORT provides the mechanism by which executable tests report PASSED, FAILED, or NOT APPLICABLE results. It also provides a set of identity functions used to defeat some compiler optimizations allowed by the Ada Standard that would circumvent a test objective. The procedure CHECK_FILE is used to check the contents of text files written by some of the Class C tests for chapter 14 of the Ada Standard. The operation of REPORT and CHECK_FILE is checked by a set of executable tests. These tests produce messages that are examined to verify that the units are operating correctly. If these units are not operating correctly, then the validation is not attempted.

The text of the tests in the ACVC follow conventions that are intended to ensure that the tests are reasonably portable without modification. For example, the tests make use of only the basic set of 55 characters, contain lines with a maximum length of 72 characters, use small numeric values, and place features that may not be supported by all implementations in separate tests. However, some tests contain values that require the test to be customized according to implementation-specific values--for example, an illegal file name. A list of the values used for this validation is provided in Appendix C.

A compiler must correctly process each of the tests in the suite and demonstrate conformity to the Ada Standard by either meeting the pass criteria given for the test or by showing that the test is inapplicable to the implementation. The applicability of a test to an implementation is considered each time the implementation is validated. A test that is inapplicable for one validation is not necessarily inapplicable for a subsequent validation. Any test that was determined to contain an illegal language construct or an erroneous language construct is withdrawn from the ACVC and, therefore, is not used in testing a compiler. The tests withdrawn at the time of this validation are given in Appendix D.

CHAPTER 2

CONFIGURATION INFORMATION

2.1 CONFIGURATION TESTED

The candidate compilation system for this validation was tested under the following configuration:

Compiler: VAda-110-03406, Version V5.51

ACVC Version: 1.9

Certificate Number: 880606W1.09083

Host Computer:

Machine:	DEC MicroVAX II
Operating System:	VMS, Version 4.7
Memory Size:	11 MB

Target Computer:

Machine:	Fairchild 9450 under Tektronix emulation
Operating System:	None
Memory Size:	64K words

2.2 IMPLEMENTATION CHARACTERISTICS

One of the purposes of validating compilers is to determine the behavior of a compiler in those areas of the Ada Standard that permit implementations to differ. Class D and E tests specifically check for such implementation differences. However, tests in other classes also characterize an implementation. The tests demonstrate the following characteristics:

- Capacities.

  The compiler correctly processes tests containing loop statements nested to 65 levels, block statements nested to 65 levels, and recursive procedures separately compiled as subunits nested to 17 levels. It correctly processes a compilation containing 723 variables in the same declarative part. (See tests D55A03A..H (8 tests), D56001B, D64005E..G (3 tests), and D29002K.)

- Universal integer calculations.

  An implementation is allowed to reject universal integer calculations having values that exceed SYSTEM.MAX_INT. This implementation processes 64 bit integer calculations. (See tests D4A002A, D4A002B, D4A004A, and D4A004B.)

- Predefined types.

  This implementation supports the additional predefined types LONG_INTEGER and LONG_FLOAT in the package STANDARD. (See tests B86001C and B86001D.)

- Based literals.

  An implementation is allowed to reject a based literal with a value exceeding SYSTEM.MAX_INT during compilation, or it may raise NUMERIC_ERROR or CONSTRAINT_ERROR during execution. This implementation raises NUMERIC_ERROR during execution. (See test E24101A.)

- Expression evaluation.

  Apparently the values of default initialization expressions for record components are checked for belonging to a component's subtype as each expression is evaluated. (See test C32117A.)

  Assignments for subtypes are performed with the same precision as the base type. (See test C35712B.)

This implementation uses no extra bits for extra precision. This implementation uses all extra bits for extra range. (See test C35903A.)

Sometimes NUMERIC_ERROR is raised when an integer literal operand in a comparison or membership test is outside the range of the base type. (See test C45232A.)

Apparently NUMERIC_ERROR is raised when a literal operand in a fixed-point comparison or membership test is outside the range of the base type. (See test C45252A.)

Apparently underflow is gradual. (See tests C45524A..Z.)

- Rounding.

The method used for rounding to integer is apparently round away from zero. (See tests C46012A..Z.)

The method used for rounding to longest integer is apparently round away from zero. (See tests C46012A..Z.)

The method used for rounding to integer in static universal real expressions is apparently round to even. (See test C4A014A.)

- Array types.

An implementation is allowed to raise NUMERIC_ERROR or CONSTRAINT_ERROR for an array having a 'LENGTH that exceeds STANDARD.INTEGER'LAST and/or SYSTEM.MAX_INT. For this implementation:

Apparently, the declaration of an array type or subtype declaration with more than SYSTEM.MAX_INT components raises no exception, except in the case of a two-dimensional array type where the second dimension is greater than SYSTEM.MAX_INT, in which case NUMERIC_ERROR is raised. (See test C36003A.)

NUMERIC_ERROR is raised when 'LENGTH is applied to an array type with INTEGER'LAST + 2 components. (See test C36202A.)

NUMERIC_ERROR is raised when 'LENGTH is applied to an array type with SYSTEM.MAX_INT + 2 components. (See test C36202B.)

A packed BOOLEAN array having a 'LENGTH exceeding INTEGER'LAST raises NUMERIC_ERROR when the array objects are sliced. (See test C52103X.)

A packed two-dimensional BOOLEAN array with more than INTEGER'LAST components raises CONSTRAINT_ERROR when the length of a dimension is calculated and exceeds INTEGER'LAST. (See test C52104Y.)

A null array with one dimension of length greater than INTEGER'LAST may raise NUMERIC_ERROR or CONSTRAINT_ERROR either when declared or assigned. Alternatively, an implementation may accept the declaration. However, lengths must match in array slice assignments. This implementation raises no exception. (See test E52103Y.)

In assigning one-dimensional array types, the expression appears to be evaluated in its entirety before CONSTRAINT_ERROR is raised when checking whether the expression's subtype is compatible with the target's subtype. In assigning two-dimensional array types, the expression does not appear to be evaluated in its entirety before CONSTRAINT_ERROR is raised when checking whether the expression's subtype is compatible with the target's subtype. (See test C52013A.)

. Discriminated types.

During compilation, an implementation is allowed to either accept or reject an incomplete type with discriminants that is used in an access type definition with a compatible discriminant constraint. This implementation accepts such subtype indications. (See test E38104A.)

In assigning record types with discriminants, the expression appears to be evaluated in its entirety before CONSTRAINT_ERROR is raised when checking whether the expression's subtype is compatible with the target's subtype. (See test C52013A.)

. Aggregates.

In the evaluation of a multi-dimensional aggregate, all choices appear to be evaluated before checking against the index type. (See tests C43207A and C43207B.)

In the evaluation of an aggregate containing subaggregates, all choices are evaluated before being checked for identical bounds. (See test E43212B.)

All choices are evaluated before CONSTRAINT_ERROR is raised if a bound in a nonnull range of a nonnull aggregate does not belong to an index subtype. (See test E43211B.)

. Representation clauses.

An implementation might legitimately place restrictions on representation clauses used by some of the tests. If a representation clause is used by a test in a way that violates a restriction, then the implementation must reject it.

For this implementation:

- Enumeration representation clauses containing noncontiguous values for enumeration types other than character and boolean types are supported. (See tests C35502I..J, C35502M..N, and A39005F.)

- Enumeration representation clauses containing noncontiguous values for character types are supported. (See tests C35507I..J, C35507M..N, and C55B16A.)

- Enumeration representation clauses for boolean types containing representational values other than (FALSE => 0, TRUE => 1) are supported. (See tests C35508I..J and C35508M..N.)

- Length clauses with SIZE specifications for enumeration types are supported. (See test A39005B.)

- Length clauses with STORAGE_SIZE specifications for access types are supported. (See tests A39005C and C87B62B.)

- Length clauses with STORAGE_SIZE specifications for task types are supported. (See tests A39005D and C87B62D.)

- Length clauses with SMALL specifications are supported. (See tests A39005E and C87B62C.)

- Record representation clauses are supported, though such a clause applied to a component of a composite type will not cause that component to be packed into the space required. In this case an explicit representation clause must be given for the component type. (See test A39005G.)

- Length clauses with SIZE specifications for derived integer types are supported. (See test C87B62A.)

- Pragmas.

  The pragma INLINE is supported for both procedures and functions. (See tests LA3004A, LA3004B, EA3004C, EA3004D, CA3004E, and CA3004F.)

- Input/output.

  The package SEQUENTIAL_IO can be instantiated with unconstrained array types and record types with discriminants without defaults. (See tests AE2101C, EE2201D, and EE2201E.)

The package DIRECT_IO can be instantiated with unconstrained array types and record types with discriminants without defaults. (See tests AE2101H, EE2401D, and EE2401G.)

The packages DIRECT_IO and SEQUENTIAL_IO do not support the procedure DELETE and any call to that procedure results in the raising of USE_ERROR. (See test CE2106A.)

The procedure CREATE in packages DIRECT_IO and SEQUENTIAL_IO does not support the creation of temporary files by use of a null string for NAME. Such calls cause the NAME_ERROR exception to be raised. (See tests CE2102C..K.)

Modes IN_FILE and OUT_FILE are supported for SEQUENTIAL_IO. (See test CE2102A.)

Modes IN_FILE, OUT_FILE, and INOUT_FILE are supported for DIRECT_IO. (See tests CE2102B and CE2111B.)

RESET is supported, but DELETE is not supported for SEQUENTIAL_IO and DIRECT_IO. (See tests CE2103A and CE2103B.)

Dynamic creation of files is supported for SEQUENTIAL_IO and DIRECT_IO. (See tests CE2106A and CE2106B.)

Overwriting to a sequential file truncates the file to last element written. (See test CE2208B.)

An existing text file can be opened in OUT_FILE mode, can be created in OUT_FILE mode, and can be created in IN_FILE mode. (See test EE3102C.)

More than one internal file can be associated with each external file for text I/O for reading only. (See tests CE3111A..E (5 tests), CE3114B, and CE3115A.)

More than one internal file can be associated with each external file for sequential I/O for reading only. (See tests CE2107A..D (4 tests), CE2110B, and CE2111D.)

More than one internal file can be associated with each external file for direct I/O for reading only. (See tests CE2107F..I (4 tests), CE2110B, and CE2111H.)

An internal sequential access file and an internal direct access file cannot be associated with a single external file for writing. (See test CE2107E.)

An external file associated with more than one internal file cannot be deleted for SEQUENTIAL_IO, DIRECT_IO, and TEXT_IO. (See test CE2110B.)

- Generics.

  Generic subprogram declarations and bodies can be compiled in separate compilations. (See tests CA1012A and CA2009F.)

  Generic package declarations and bodies can be compiled in separate compilations. (See tests CA2009C, BC3204C, and BC3205D.)

  Generic unit bodies and their subunits can be compiled in separate compilations. (See test CA3011A.)

CHAPTER 3

TEST INFORMATION

3.1 TEST RESULTS

Version 1.9 of the ACVC comprises 3122 tests. When this compiler was tested, 27 tests had been withdrawn because of test errors. The AVF determined that 480 tests were inapplicable to this implementation. All inapplicable tests were processed during validation testing except for 285 executable tests that use floating-point precision exceeding that supported by the implementation. Modifications to the code, processing, or grading for 27 tests were required to successfully demonstrate the test objective. (See section 3.6.)

The AVF concludes that the testing results demonstrate acceptable conformity to the Ada Standard.

3.2 SUMMARY OF TEST RESULTS BY CLASS

RESULT	TEST CLASS						TOTAL
	A	B	C	D	E	L	
Passed	108	1048	1382	17	14	46	2615
Inapplicable	2	3	471	0	4	0	480
Withdrawn	3	2	21	0	1	0	27
TOTAL	113	1053	1874	17	19	46	3122

3.3 SUMMARY OF TEST RESULTS BY CHAPTER

RESULT	CHAPTER													TOTAL
	2	3	4	5	6	7	8	9	10	11	12	13	14	
Passed	184	469	490	245	166	98	141	326	137	36	234	3	86	2615
Inapplicable	20	103	184	3	0	0	2	1	0	0	0	0	167	480
Withdrawn	2	14	3	0	0	1	2	0	0	0	2	1	2	27
TOTAL	206	586	677	248	166	99	145	327	137	36	236	4	255	3122

3.4 WITHDRAWN TESTS

The following 27 tests were withdrawn from ACVC Version 1.9 at the time of this validation:

B28003A	E28005C	C34004A	C35502P	A35902C
C35904A	C35904B	C35A03E	C35A03R	C37213H
C37213J	C37215C	C37215E	C37215G	C37215H
C38102C	C41402A	C45332A	C45614C	A74106C
C85018B	C87B04B	CC1311B	BC3105A	AD1A01A
CE2401H	CE3208A			

See Appendix D for the reason that each of these tests was withdrawn.

3.5 INAPPLICABLE TESTS

Some tests do not apply to all compilers because they make use of features that a compiler is not required by the Ada Standard to support. Others may depend on the result of another test that is either inapplicable or withdrawn. The applicability of a test to an implementation is considered each time a validation is attempted. A test that is inapplicable for one validation attempt is not necessarily inapplicable for a subsequent attempt. For this validation attempt, 480 tests were inapplicable for the reasons indicated:

- C35702A uses SHORT_FLOAT which is not supported by this implementation.

- A39005G uses a record representation clause which is not supported by this compiler.

- The following tests use SHORT_INTEGER, which is not supported by this compiler:

C45231B	C45304B	C45502B	C45503B	C45504B
C45504E	C45611B	C45613B	C45614B	C45631B
C45632B	B52004E	C55B07B	B55B09D	

- C45531M, C45531N, C45532M, and C45532N use fine 48-bit fixed-point base types which are not supported by this compiler.

- C45531O, C45531P, C45532O, and C45532P use coarse 48-bit fixed-point base types which are not supported by this compiler.

- C45231D and B86001D require a macro substitution for any predefined numeric type other than INTEGER, SHORT_INTEGER, LONG_INTEGER, FLOAT, SHORT_FLOAT, and LONG_FLOAT. This compiler does not support any such types.

- C86001F redefines package SYSTEM, but TEXT_IO is made obsolete by this new definition in this implementation and the test cannot be executed since the package REPORT is dependent on the package TEXT_IO.

- C96005B requires the range of type DURATION to be different from those of its base type; in this implementation they are the same.

- The following 285 tests require a floating-point accuracy that exceeds the maximum of 9 digits supported by this implementation:

C24113F..Y (20 tests)	C35705F..Y (20 tests)
C35706F..Y (20 tests)	C35707F..Y (20 tests)
C35708F..Y (20 tests)	C35802F..Z (21 tests)
C45241F..Y (20 tests)	C45321F..Y (20 tests)
C45421F..Y (20 tests)	C45521F..Z (21 tests)
C45524F..Z (21 tests)	C45621F..Z (21 tests)
C45641F..Y (20 tests)	C46012F..Z (21 tests)

- CE2107B..E,G..I (7 tests), CE2110B, CE2111D, CE2111H, CE3111B..E (4 tests), and CE3114B are inapplicable because multiple internal files cannot be associated with the same external file when any of the internal files are opened for writing and DELETE in SEQUENTIAL_IO or DIRECT_IO is not supported.

- The following 38 tests use DELETE in SEQUENTIAL_IO or DIRECT_IO which is not supported by this compiler:

AE3101A	CE2105A	CE2105B	CE2106A	CE2106B
CE2111C	CE2111G	CE2408A	CE3102B	CE3103A
CE3109A	CE3111A	CE3112B	CE3203A	CE3208A
CE3301B	CE3305A	CE3405A	CE3602A	CE3602B
CE3603A	CE3604A	CE3605D	CE3704E	CE3704F
CE3704M	CE3704N	CE3704O	CE3804A	CE3804B
CE3804D	CE3804E	CE3804G	CE3804K	CE3804M
CE3805A	CE3805B	CE39051		

- The following 114 tests use CREATE in SEQUENTIAL_IO or DIRECT_IO with a null string for NAME to create a temporary file:

CE2102C	CE2102D	CE2102E	CE2102F	CE2102G
CE2102H	CE2102I	CE2102J	CE2102K	CE2108A
CE2108B	CE2108C	CE2108D	CE2109A	CE2109B
CE2109C	CE2201A	CE2201B	CE2201C	CE2201F
CE2201G	CE2202A	CE2204A	CE2204B	CE2210A
CE2401A	CE2401B	CE2401C	CE2401E	CE2401F
CE2404A	CE2405B	CE2406A	CE2407A	CE2409A
CE2410A	CE3102A	CE3104A	CE3112A	CE3301A
CE3301C	CE3302A	CE3402A	CE3402B	CE3402C
CE3402D	CE3403A	CE3403B	CE3403C	CE3403E
CE3403F	CE3404A	CE3404B	CE3404C	CE3405B
CE3405C	CE3405D	CE3406A	CE3406B	CE3406C
CE3406D	CE3407A	CE3407B	CE3407C	CE3408A
CE3408B	CE3408C	CE3409A	CE3409C	CE3409D
CE3409E	CE3409F	CE3410A	CE3410C	CE3410D
CE3410E	CE3410F	CE3411A	CE3411C	CE3412A
CE3412C	CE3413A	CE3413C	CE3602C	CE3602D
CE3605A	CE3605B	CE3605C	CE3605E	CE3606A
CE3606B	CE3701A	CE3704A	CE3704B	CE3704D
CE3706D	CE3706F	CE3804C	CE3804I	CE3806A
CE3806D	CE3806E	CE3905A	CE3905B	CE3905C
CE3906A	CE3906B	CE3906C	CE3906E	CE3906F
EE2201D	EE2201E	EE2401D	EE2401G	

In addition, the support test CZ1103A attempts to use CREATE with a null string for NAME and is graded as inapplicable.

3.6 TEST, PROCESSING, AND EVALUATION MODIFICATIONS

It is expected that some tests will require modifications of code, processing, or evaluation in order to compensate for legitimate implementation behavior. Modifications are made by the AVF in cases where legitimate implementation behavior prevents the successful completion of an (otherwise) applicable test. Examples of such modifications include: adding a length clause to alter the default size of a collection; splitting a Class B test into subtests so that all errors are detected; and confirming that messages produced by an executable test demonstrate conforming behavior that wasn't anticipated by the test (such as raising one exception instead of another).

Modifications were required for 26 Class B tests and 1 Class C test.

The following Class B tests were split because syntax errors at one point resulted in the compiler not detecting other errors in the test:

B24009A	B24204A	B24204B	B24204C	B2A003A
B2A003B	B2A003C	B33301A	B37201A	B38003A
B38003B	B38009A	B38009B	B41202A	B44001A
B64001A	B67001A	B67001B	B67001C	B67001D

BC1303F	BC3005B	B91001H	B91003B	B95001A
B97102A				

Test CE1221A was split into five parts because the object code produced by the compiler was too large to execute on the target.

3.7 ADDITIONAL TESTING INFORMATION

3.7.1 Prevalidation

Prior to validation, a set of test results for ACVC Version 1.9 produced by the VAda-110-03406, Version V5.51, was submitted to the AVF by the applicant for review. Analysis of these results demonstrated that the compiler successfully passed all applicable tests, and the compiler exhibited the expected behavior on all inapplicable tests.

3.7.2 Test Method

Testing of the VAda-110-03406, Version V5.51, compiler using ACVC Version 1.9 was conducted on-site by a validation team from the AVF. The configuration consisted of a DEC MicroVAX II host operating under VMS, Version 4.7, and a bare Fairchild 9450 target.

A magnetic tape containing all tests except for withdrawn tests and tests requiring unsupported floating-point precisions was taken on-site by the validation team for processing. Tests that make use of implementation-specific values were customized before being written to the magnetic tape. Tests requiring modifications during the prevalidation testing were included in their modified form on the magnetic tape.

The contents of the magnetic tape were loaded onto a Sun Microsystems Sun-3 computer. After modifying the test name extensions to make them compatible with the system naming conventions, the test sources were copied into the test area on the MicroVAX II machine with FTP (File Transfer Protocol) over a network system implementing standard IP on Ethernet.

After the test files were loaded to disk, the full set of tests was compiled and linked on the MicroVAX II, and all executable tests were run on the Fairchild 9450. Results were transferred back to the MicroVAX II and routed to the network printer. They were then checked by the validation team.

The compiler was tested using command scripts provided by Verdix Corporation and reviewed by the validation team. The compiler was tested using all default switch settings.

The REPORT package was modified to use SIMPLE_IO, a simplified version of TEXT_IO, for all tests except those in Chapter 14. The implementation of SIMPLE_IO uses a division of TEXT_IO developed by Verdix Corporation for previous validations performed from VMS to cross-target bare machines. In this cross-target implementation, the functions of TEXT_IO are logically and physically divided into two portions which run on both the host and the target. I/O file system requests are handled by the portion running on the host; output formatting is handled by the portion running on the target. Both portions are written in Ada. For the most part, this implementation is completely transparent to the user, except that certain default file characteristics will be determined by the host operating system. A protocol has been developed to allow the target processor to make requests of the host file system by means of a daemon on the host. Any host on which the daemon is implemented can serve as the file system server for the target processor; thus this underlying implementation of TEXT_IO is independent of the host operating system.

Tests were compiled, linked, and executed (as appropriate) using a single host computer and a single target computer. Test output, compilation listings, and job logs were captured on magnetic tape and archived at the AVF. The listings examined on-site by the validation team were also archived.

3.7.3 Test Site

Testing was conducted at the facilities of Verdix Corporation in Aloha, OR and was completed on 13 June 1988.

APPENDIX A

DECLARATION OF CONFORMANCE

Verdix Corporation has submitted the following Declaration of Conformance concerning the VAda-110-03406, Version V5.51, compiler.

DECLARATION OF CONFORMANCE

Compiler Implementor: Verdix Corporation
Ada Validation Facility: Ada Validation Facility, ASD/SCEL, Wright-Patterson AFB OH 45433-6503
Ada Compiler Validation Capability (ACVC) Version: 1.9

Base Configuration

Base Compiler Name:	VAda-110-03406	Version:	V5.51
Host Architecture ISA:	DEC MicroVAX II	OS&VER #:	VMS, Version 4.7
Target Architecture ISA:	Fairchild 9450 under Tektronix emulation	OS&VER #:	(None)

Implementor's Declaration

I, the undersigned, representing Verdix Corporation, have implemented no deliberate extensions to the Ada Language Standard ANSI/MIL-STD-1815A in the compiler(s) listed in this declaration. I declare that Verdix Corporation is the owner of record of the Ada language compiler(s) listed above and, as such, is responsible for maintaining said compiler(s) in conformance to ANSI/MIL-STD-1815A. All certificates and registrations for Ada language compiler(s) listed in this declaration shall be made only in the owner's corporate name.

[illegible] Date: 6/9/88
Verdix Corporation
Gregory Burns, Project Manager

Owner's Declaration

I, the undersigned, representing Verdix Corporation, take full responsibility for implementation and maintenance of the Ada compiler(s) listed above, and agree to the public disclosure of the final Validation Summary Report. I further agree to continue to comply with the Ada trademark policy, as defined by the Ada Joint Program Office. I declare that all of the Ada language compilers listed, and their host/target performance, are in compliance with the Ada Language Standard ANSI/MIL-STD-1815A.

[illegible] Date: 6/9/88
Verdix Corporation
Gregory Burns, Project Manager

APPENDIX B

APPENDIX F OF THE Ada STANDARD

The only allowed implementation dependencies correspond to implementation-dependent pragmas, to certain machine-dependent conventions as mentioned in chapter 13 of the Ada Standard, and to certain allowed restrictions on representation clauses. The implementation-dependent characteristics of VAda-110-03406, Version V5.51, are described in the following sections, taken from Appendix F of the Ada Standard. Implementation-specific portions of the package STANDARD are also included in this appendix.

```
package STANDARD is

     ...

     type INTEGER is range -32768 .. 32767;
     type LONG_INTEGER is range -2147483648 .. 2147483647;

     type FLOAT is digits 6
                    range -1.7014118346046E+38 .. 1.70141163317805E+38;
     type LONG_FLOAT is digits 9
                    range -1.7014118346046E+38 .. 1.7014118346015E+38;

     type DURATION is delta 1.0E-03 range -2147483.647 .. 2147483.646;

     ...

end STANDARD;
```

ATTACHMENT I

APPENDIX F. Implementation-Dependent Characteristics

1. Implementation-Dependent Pragmas

1.1. INLINE_ONLY Pragma

The INLINE_ONLY pragma, when used in the same way as progma INLINE, indicates to the compiler that the subprogram must *always* be inlined. This pragma also suppresses the generation of a callable version of the routine which save code space.

1.2. BUILT_IN Pragma

The EUILT_IN pragma is used in the implementation of some predefined Ada packages, but provides no user access. It is used only to implement code bodies for which no actual Ada body can be provided, for example the MACHINE_CODE package.

1.3. SHARE_CODE Pragma

The SHARE_CODE pragma takes the name of a generic instantiation or a generic unit as the first argument and one of the identifiers TRUE or FALSE as the second argument. This pragma is only allowed immediately at the place of a declarative item in a declarative part or package specification, or after a library unit in a compilation, but before any subsequent compilation unit.

When the first argument is a generic unit the pragma applies to all instantiations of that generic. When the first argument is the name of a generic instantiation the pragma applies only to the specified instantiation, or overloaded instantiations.

If the second argument is TRUE the compiler will try to share code generated for a generic instantiation with code generated for other instantiations of the same generic. When the second argument is FALSE each instantiation will get a unique copy of the generated code. The extent to which code is shared between instantiations depends on this pragma and the kind of generic formal parameters declared for the generic unit.

The name pragma SHARE_BODY is also recognized by the implementation and has the same effect as SHARE_CODE. It is included for compatability with earlier versions of VADS.

1.4. NO_IMAGE Pragma

The pragma suppresses the generation of the image array used for the IMAGE attribute of enumeration types. This eliminates the overhead required to store the array in the executable image.

1.5. EXTERNAL_NAME Pragma

The EXTERNAL_NAME pragma takes the name of a subprogram or variable defined in Ada and allows the user to specify a different external name that may be used to reference the entity from other languages. The pragma is allowed at the place of a declarative item in a package specification and must apply to an object declared earlier in the same package specification.

1.6. INTERFACE_OBJECT Pragma

The INTERFACE_OBJECT pragma takes the name of a a variable defined in another language and allows it to be referenced directly in Ada. The pragma will replace all occurrences of the variable name with an external reference to the second, link_argument. The pragma is allowed at the place of a declarative item in a package specification and must apply to an object declared earlier in the same package specification. The object must be declared as a scalar or an access type. The object *cannot* be any of the following:

a loop variable,
a constant,
an initialized variable,
an array, or
a record.

1.7. IMPLICIT_CODE Pragma

Takes one of the identifiers ON or OFF as the single argument. This pragma is only allowed within a machine code procedure. It specifies that implicit code generated by the compiler be allowed or disallowed. A warning is issued if OFF is used and any implicit code needs to be generated. The default is ON.

2. Implementation of Predefined Pragmas

2.1. CONTROLLED

This pragma is recognized by the implementation but has no effect.

2.2. ELABORATE

This pragma is implemented as described in Appendix B of the Ada RM.

2.3. INLINE

This pragma is implemented as described in Appendix B of the Ada RM.

2.4. INTERFACE

This pragma supports calls to 'C' and FORTRAN functions. The Ada subprograms can be either functions or procedures. The types of parameters and the result type for functions must be scalar, access or the predefined type ADDRESS in SYSTEM. An optional third argument overrides the default link name. All parameters must have mode IN. Record and array objects can be passed by reference using the ADDRESS attribute.

2.5. LIST

This pragma is implemented as described in Appendix B of the Ada RM.

2.6. MEMORY_SIZE

This pragma is recognized by the implementation. The implementation does not allow SYSTEM to be modified by means of pragmas, the SYSTEM package must be recompiled.

2.7. OPTIMIZE

This pragma is recognized by the implementation but has no effect.

2.8. PACK

This pragma will cause the compiler to choose a non-aligned representation for composite types. It will not causes objects to be packed at the bit level.

2.5. PAGE

This pragma is implemented as described in Appendix B of the Ada RM.

2.10. PRIORITY

This pragma is implemented as described in Appendix B of the Ada RM.

2.11. SHARED

This pragma is recognized by the implementation but has no effect.

2.12. STORAGE_UNIT

This pragma is recognized by the implementation. The implementation does not allow SYSTEM to be modified by means of pragmas, the SYSTEM package must be recompiled.

2.13. SUPPRESS

This pragma is implemented as described, except that RANGE_CHECK and DIVISION_CHECK cannot be supressed.

2.14. SYSTEM_NAME

This pragma is recognized by the implementation. The implementation does not allow SYSTEM to be modified by means of pragmas, the SYSTEM package must be recompiled.

3. Implementation-Dependent Attributes

3.1. P'REF

For a prefix that denotes an object, a program unit, a label, or an entry:

This attribute denotes the effective address of the first of the storage units allocated to P. For a subprogram, package, task unit, or label, it refers to the address of the machine code associated with the corresponding body or statement. For an entry for which an address clause has been given, it refers to the corresponding hardware interrupt. The attribute is of the type OPERAND defined in the package MACHINE_CODE. The attribute is only allowed within a machine code procedure.

(For a package, task unit, or entry, the 'REF attribute is not supported.)

4. Specification Of Package SYSTEM

```
package SYSTEM
is
        type NAME is ( mi750a );

        SYSTEM_NAME                     : constant NAME := mi750a;
        EXTENDED_MEMORY       : BOOLEAN := FALSE;
        STORAGE_UNIT          : constant := 16;
        MEMORY_SIZE                     : constant := 2097152;

        -- System-Dependent Named Numbers

        MIN_INT                         : constant := -2_147_483_648;
        MAX_INT                         : constant := 2_147_483_647;
        MAX_DIGITS                      : constant := 9;
        MAX_MANTISSA          : constant := 31;
        FINE_DELTA                      : constant := 2.0**(-31);
        TICK                            : constant := 0.01;

        -- Other System-dependent Declarations

        subtype PRIORITY is INTEGER range 0 .. 99;

        MAX_REC_SIZE : integer := 1*1024;
        type SHORT_ADDRESS is private;
        type ADDRESS is private;
        NO_ADDR : constant ADDRESS;
        NO_SHORT_ADDR : constant SHORT_ADDRESS;
        subtype SEGMENT is INTEGER range 0 .. INTEGER'LAST;
        function PHYSICAL_ADDRESS(I: INTEGER) return ADDRESS;
        function ADDR_GT(A, B: ADDRESS) return BOOLEAN;
        function ADDR_LT(A, B: ADDRESS) return BOOLEAN;
        function ADDR_GE(A, B: ADDRESS) return BOOLEAN;
        function ADDR_LE(A, B: ADDRESS) return BOOLEAN;
        function ADDR_DIFF(A, B: ADDRESS) return INTEGER;
        function INCR_ADDR(A: ADDRESS; INCR: INTEGER) return ADDRESS;
        function DECR_ADDR(A: ADDRESS; DECR: INTEGER) return ADDRESS;

        function ">"(A, B: ADDRESS) return BOOLEAN renames ADDR_GT;
        function "<"(A, B: ADDRESS) return BOOLEAN renames ADDR_LT;
        function ">="(A, B: ADDRESS) return BOOLEAN renames ADDR_GE;
        function "<="(A, B: ADDRESS) return BOOLEAN renames ADDR_LE;
        function "-"(A, B: ADDRESS) return INTEGER renames ADDR_DIFF;
        function "+"(A: ADDRESS; INCR: INTEGER) return ADDRESS renames INCR_ADDR;
        function "-"(A: ADDRESS; DECR: INTEGER) return ADDRESS renames DECR_ADDR;

        function OFFSET_OF(A: ADDRESS) return SHORT_ADDRESS;
        function SEGMENT_OF(A: ADDRESS) return SEGMENT;
        function SEGMENT_OF return SEGMENT;
        function MAKE_ADDRESS(A: SHORT_ADDRESS; SEG: SEGMENT) return ADDRESS;

        function PHYSICAL_ADDRESS(I: LONG_INTEGER) return SHORT_ADDRESS;
        function ADDR_GT(A, B: SHORT_ADDRESS) return BOOLEAN;
        function ADDR_LT(A, B: SHORT_ADDRESS) return BOOLEAN;
        function ADDR_GE(A, B: SHORT_ADDRESS) return BOOLEAN;
        function ADDR_LE(A, B: SHORT_ADDRESS) return BOOLEAN;
        function ADDR_DIFF(A, B: SHORT_ADDRESS) return INTEGER;
        function INCR_ADDR(A: SHORT_ADDRESS; INCR: INTEGER) return SHORT_ADDRESS;
        function DECR_ADDR(A: SHORT_ADDRESS; DECR: INTEGER) return SHORT_ADDRESS;

        function ">"(A, B: SHORT_ADDRESS) return BOOLEAN renames ADDR_GT;
        function "<"(A, B: SHORT_ADDRESS) return BOOLEAN renames ADDR_LT;
        function ">="(A, B: SHORT_ADDRESS) return BOOLEAN renames ADDR_GE;
        function "<="(A, B: SHORT_ADDRESS) return BOOLEAN renames ADDR_LE;
        function "-"(A, B: SHORT_ADDRESS) return INTEGER renames ADDR_DIFF;
        function "+"(A: SHORT_ADDRESS; INCR: INTEGER) return SHORT_ADDRESS
                renames INCR_ADDR;
        function "-"(A: SHORT_ADDRESS; DECR: INTEGER) return SHORT_ADDRESS
                renames DECR_ADDR;

        pragma inline(ADDR_GT);
        pragma inline(ADDR_LT);
        pragma inline(ADDR_GE);
        pragma inline(ADDR_LE);
        pragma inline(ADDR_DIFF);
        pragma inline(INCR_ADDR);
        pragma inline(DECR_ADDR);
        pragma inline(OFFSET_OF);
        pragma inline(SEGMENT_OF);
        pragma inline(MAKE_ADDRESS);
        pragma inline(PHYSICAL_ADDRESS);

private

        type ADDRESS is new integer;
        type SHORT_ADDRESS is new address;
        for ADDRESS'size use 16;
        for SHORT_ADDRESS'size use 16;
        NO_ADDR                        : constant ADDRESS := 0;
        NO_SHORT_ADDR        : constant SHORT_ADDRESS := 0;

end SYSTEM;
```

5. Restrictions On Representation Clauses

5.1. Pragma PACK

Array components less than STORAGE_UNIT bits are packed to the next highest power of 2 bits. Objects and larger components are packed to the nearest whole STORAGE_UNIT. In the absence of pragma PACK record components are padded so as to provide for efficient access by the target hardware, pragma PACK applied to a record eliminated the padding where possible. Pragma PACK has no other effect on the storage allocate for record components a record representation is required.

5.2. Record Representation Clauses

For scalar types a represenation clause will pack to the number of bits required to represent the range of the subtype. A record representation applied to a composite type will not cause the object to be packed to fit in the space required. An explicit representation clause must be given for the component type. An error will be issued if there is unsufficient space allocated.

In a record representation clause, a component clause for a component of a composite type may not specify a smaller size than would otherwise be occupied by the component. In addition, a subcomponent type will not be packed unless an explicit representation specification or pragma packed is applied to the type.

5.3. Address Clauses

Address clauses are supported for variables and constants.

5.4. Interrupts

Interupt entries are not supported.

5.5. Representation Attributes

The ADDRESS attribute is not supported for the following entities:

Packages
Tasks
Labels
Entries

5.6. Machine Code Insertions

Machine code insertions are supported.

The general definition of the package MACHINE_CODE provides an assembly language interface for the target machine. It provides the necessary record type(s) needed in the code statement, an enumeration type of all the opcode mneumonics, a set of register definitions, and a set of addressing mode functions.

The general syntax of a machine code statement is as follows:

CODE_*n*'(*opcode*, *operand* {, *operand*});

where *n* indicates the number of operands in the aggregate.

A special case arises for a variable number of operands. The operands are listed within a subaggregate. The format is as follows:

CODE_N'(*opcode*, (*operand* {, *operand*}));

For those opcodes that require no operands, named notation must be used (cf. RM 4.3(4)).

CODE_0'(op => *opcode*);

The *opcode* must be an enumeration literal (i.e. it cannot be an object, attribute, or a rename).

An *operand* can only be an entity defined in MACHINE_CODE or the 'REF attribute.

The arguments to any of the functions defined in MACHINE_CODE must be static expressions, string literals, or the functions defined in MACHINE_CODE. The 'REF attribute may not be used as an argument in any of these functions.

Inline expansion of machine code procedures is supported.

6. Conventions for Implementation-generated Names

There are no implementation-generated names.

7. Interpretation of Expressions in Address Clauses

Address clauses are supported for constants and variables.

8. Restrictions on Unchecked Conversions

None.

9. Restrictions on Unchecked Deallocations

None.

10. Implementation Characteristics of I/O Packages

Instantiations of DIRECT_IO use the value MAX_REC_SIZE as the record size (expressed in STORAGE_UNITS) when the size of ELEMENT_TYPE exceeds that value. For example for unconstrained arrays such as string where ELEMENT_TYPE'SIZE is very large, MAX_REC_SIZE is used instead. MAX_RECORD_SIZE is defined in SYSTEM and can be changed by a program before instantiating DIRECT_IO to provide an upper limit on the record size. In any case the maximum size supported is 1024 x 1024 x STORAGE_UNIT bits. DIRECT_IO will raise USE_ERROR if MAX_REC_SIZE exceeds this absolute limit.

Instantiations of SEQUENTIAL_IO use the value MAX_REC_SIZE as the record size (expressed in STORAGE_UNITS) when the size of ELEMENT_TYPE exceeds that value. For example for unconstrained arrays such as string where ELEMENT_TYPE'SIZE is very large, MAX_REC_SIZE is used instead. MAX_RECORD_SIZE is defined in SYSTEM and can be changed by a program before instantiating INTEGER_IO to provide an upper limit on the record size. SEQUENTIAL_IO imposes no limit on MAX_REC_SIZE.

11. Implementation Limits

The following limits are actually enforced by the implementation. It is not intended to imply that resources up to or even near these limits are available to every program.

11.1. Line Length

The implementation supports a maximum line length of 500 characters including the end of line character.

11.2. Record and Array Sizes

The maximum size of a statically sized array type is 32,767 x STORAGE_UNITS. The maximum size of a statically sized record type is 32,767 x STORAGE_UNITS. A record type or array type declaration that exceeds these limits will generate a warning message.

11.3. Default Stack Size for Tasks

In the absence of an explicit STORAGE_SIZE length specification every task except the main program is allocated a fixed size stack of 400 STORAGE_UNITS. This is the value returned by T'STORAGE_SIZE for a task type T.

11.4. Default Collection Size

In the absence of an explicit STORAGE_SIZE length attribute the default collection size for an access type is 100 times the number of STORAGE_UNITS needed by the accessed object. This is the value returned by T'STORAGE_SIZE for an access type T.

11.5. Limit on Declared Objects

There is an absolute limit of 32,767 x STORAGE_UNITS for objects declared statically within a compilation unit. If this value is exceeded the compiler will terminate the compilation of the unit with a FATAL error message.

11.6. External Files cannot be Deleted

The Tektronix 8540 environment does not support deletion of external files. The procedure DELETE raises USE_ERROR as decribed in RM 14.2.1(13).

11.7. Temporary Files cannot be Created

The Tektronix 8540 environment does not support creation of temporary files. Calls to procedure CREATE with a null string for NAME specifies a temporary file as described in RM 14.2.1(3). Such calls cause NAME_ERROR to be raised.

APPENDIX C

TEST PARAMETERS

Certain tests in the ACVC make use of implementation-dependent values, such as the maximum length of an input line and invalid file names. A test that makes use of such values is identified by the extension .TST in its file name. Actual values to be substituted are represented by names that begin with a dollar sign. A value must be substituted for each of these names before the test is run. The values used for this validation are given below.

Name and Meaning	Value
$BIG_ID1 Identifier the size of the maximum input line length with varying last character.	(1..498 => 'A', 499 => '1')
$BIG_ID2 Identifier the size of the maximum input line length with varying last character.	(1..498 => 'A', 499 => '2')
$BIG_ID3 Identifier the size of the maximum input line length with varying middle character.	(1..249 | 251..499 => 'A', 250 => '3')
$BIG_ID4 Identifier the size of the maximum input line length with varying middle character.	(1..249 | 251..499 => 'A', 250 => '4')
$BIG_INT_LIT An integer literal of value 298 with enough leading zeroes so that it is the size of the maximum line length.	(1..496 => '0', 497..499 => "298")

TEST PARAMETERS

Name and Meaning	Value
$BIG_REAL_LIT A universal real literal of value 690.0 with enough leading zeroes to be the size of the maximum line length.	(1..493 => '0', 494..499 => "69.0E1")
$BIG_STRING1 A string literal which when catenated with BIG_STRING2 yields the image of BIG_ID1.	(1..250 => 'A')
$BIG_STRING2 A string literal which when catenated to the end of BIG_STRING1 yields the image of BIG_ID1.	(1..248 => 'A', 249 => '1')
$BLANKS A sequence of blanks twenty characters less than the size of the maximum line length.	(1..479 => ' ')
$COUNT_LAST A universal integer literal whose value is TEXT_IO.COUNT'LAST.	2147483647
$FIELD_LAST A universal integer literal whose value is TEXT_IO.FIELD'LAST.	2147483647
$FILE_NAME_WITH_BAD_CHARS An external file name that either contains invalid characters or is too long.	/illegal/file_name/2{]$%2102C.DAT
$FILE_NAME_WITH_WILD_CARD_CHAR An external file name that either contains a wild card character or is too long.	/illegal/file_name/CE2102C*.DAT
$GREATER_THAN_DURATION A universal real literal that lies between DURATION'BASE'LAST and DURATION'LAST or any value in the range of DURATION.	100_000.0

Name and Meaning	Value
$GREATER_THAN_DURATION_BASE_LAST A universal real literal that is greater than DURATION'BASE'LAST.	10_000_000.0
$ILLEGAL_EXTERNAL_FILE_NAME1 An external file name which contains invalid characters.	/no/such/directory /ILLEGAL_EXTERNAL_FILE_NAME1
$ILLEGAL_EXTERNAL_FILE_NAME2 An external file name which is too long.	/no/such/directory /ILLEGAL_EXTERNAL_FILE_NAME2
$INTEGER_FIRST A universal integer literal whose value is INTEGER'FIRST.	-2147483648
$INTEGER_LAST A universal integer literal whose value is INTEGER'LAST.	2147483647
$INTEGER_LAST_PLUS_1 A universal integer literal whose value is INTEGER'LAST + 1.	2147483648
$LESS_THAN_DURATION A universal real literal that lies between DURATION'BASE'FIRST and DURATION'FIRST or any value in the range of DURATION.	-100_000.0
$LESS_THAN_DURATION_BASE_FIRST A universal real literal that is less than DURATION'BASE'FIRST.	-10_000_000.0
$MAX_DIGITS Maximum digits supported for floating-point types.	9
$MAX_IN_LEN Maximum input line length permitted by the implementation.	499
$MAX_INT A universal integer literal whose value is SYSTEM.MAX_INT.	2147483647
$MAX_INT_PLUS_1 A universal integer literal whose value is SYSTEM.MAX_INT+1.	2147483648

TEST PARAMETERS

Name and Meaning	Value
$MAX_LEN_INT_BASED_LITERAL A universal integer based literal whose value is 2#11# with enough leading zeroes in the mantissa to be MAX_IN_LEN long.	(1..2 => "2:", 3..496 => '0', 497..499 => "11:")
$MAX_LEN_REAL_BASED_LITERAL A universal real based literal whose value is 16:F.E: with enough leading zeroes in the mantissa to be MAX_IN_LEN long.	(1..3 => "16:", 4..495 => '0', 496..499 => "F.E:")
$MAX_STRING_LITERAL A string literal of size MAX_IN_LEN, including the quote characters.	(1 => '"', 2..498 => 'A', 499 => '"')
$MIN_INT A universal integer literal whose value is SYSTEM.MIN_INT.	-2147483648
$NAME A name of a predefined numeric type other than FLOAT, INTEGER, SHORT_FLOAT, SHORT_INTEGER, LONG_FLOAT, or LONG_INTEGER.	[There is none.]
$NEG_BASED_INT A based integer literal whose highest order nonzero bit falls in the sign bit position of the representation for SYSTEM.MAX_INT.	16#FFFFFFFD#

APPENDIX D

WITHDRAWN TESTS

Some tests are withdrawn from the ACVC because they do not conform to the Ada Standard. The following 27 tests had been withdrawn at the time of validation testing for the reasons indicated. A reference of the form "AI-ddddd" is to an Ada Commentary.

- B28003A: A basic declaration (line 36) incorrectly follows a later declaration.
- E28005C: This test requires that "PRAGMA LIST (ON);" not appear in a listing that has been suspended by a previous "PRAGMA LIST (OFF);"; the Ada Standard is not clear on this point, and the matter will be reviewed by the AJPO.
- C34004A: The expression in line 168 yields a value outside the range of the target type T, but there is no handler for CONSTRAINT_ERROR.
- C35502P: The equality operators in lines 62 and 69 should be inequality operators.
- A35902C: The assignment in line 17 of the nominal upper bound of a fixed-point type to an object raises CONSTRAINT_ERROR, for that value lies outside of the actual range of the type.
- C35904A: The elaboration of the fixed-point subtype on line 28 wrongly raises CONSTRAINT_ERROR, because its upper bound exceeds that of the type.
- C35904B: The subtype declaration that is expected to raise CONSTRAINT_ERROR when its compatibility is checked against that of various types passed as actual generic parameters, may, in fact, raise NUMERIC_ERROR or CONSTRAINT_ERROR for reasons not anticipated by the test.

- C35A03E and C35A03R: These tests assume that attribute 'MANTISSA returns 0 when applied to a fixed-point type with a null range, but the Ada Standard does not support this assumption.

- C37213H: The subtype declaration of SCONS in line 100 is incorrectly expected to raise an exception when elaborated.

- C37213J: The aggregate in line 451 incorrectly raises CONSTRAINT_ERROR.

- C37215C, C37215E, C37215G, and C37215H: Various discriminant constraints are incorrectly expected to be incompatible with type CONS.

- C38102C: The fixed-point conversion on line 23 wrongly raises CONSTRAINT_ERROR.

- C41402A: The attribute 'STORAGE_SIZE is incorrectly applied to an object of an access type.

- C45332A: The test expects that either an expression in line 52 will raise an exception or else MACHINE_OVERFLOWS is FALSE. However, an implementation may evaluate the expression correctly using a type with a wider range than the base type of the operands, and MACHINE_OVERFLOWS may still be TRUE.

- C45614C: The function call of IDENT_INT in line 15 uses an argument of the wrong type.

- A74106C, C85018B, C87B04B, and CC1311B: A bound specified in a fixed-point subtype declaration lies outside of that calculated for the base type, raising CONSTRAINT_ERROR. Errors of this sort occur at lines 37 & 59, 142 & 143, 16 & 48, and 252 & 253 of the four tests, respectively.

- BC3105A: Lines 159 through 168 expect error messages, but these lines are correct Ada.

- AD1A01A: The declaration of subtype SINT3 raises CONSTRAINT_ERROR for implementations which select INT'SIZE to be 16 or greater.

- CE2401H: The record aggregates in lines 105 and 117 contain the wrong values.

- CE3208A: This test expects that an attempt to open the default output file (after it was closed) with mode IN_FILE raises NAME_ERROR or USE_ERROR; by Commentary AI-00048, MODE_ERROR should be raised.

SUPPLEMENTARY

INFORMATION

ERRATA DTIC# A204 424

AD-A-204 424

Ada Compiler Validation Summary Report:

Compiler Name: VAda-110-03406, Version V5.51

Certificate Number: 880606W1.09083

Host:	Target:
DEC MicroVAX II under VMS, Version 4.7	Fairchild 9450 under Tektronix emulation

Testing Completed 13 June 1988 Using ACVC 1.9

This report has been reviewed and is approved.

[illegible]

Ada Validation Facility
Steven P. Wilson
Technical Director
ASD/SCEL
Wright-Patterson AFB OH 45433-6503

[illegible]

Ada Validation Organization
Dr. John F. Kramer
Institute for Defense Analyses
Alexandria VA 22311

Virginia L. Castor

Ada Joint Program Office
Virginia L. Castor
Director
Department of Defense
Washington DC 20301